Journal of Universal Computer Science, vol. 20, no. 4 (2014), 534-560
submitted: 21/1/14, accepted: 28/3/14, appeared: 1/4/14 © J.UCS

An Approach for Mapping Domain-Specific AOM Applications to a General Model

Patricia Matsumoto
(Instituto Tecnológico de Aeronáutica, São José dos Campos, Brazil
patricia.matsumoto@gmail.com)

Eduardo Guerra
(Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brazil
eduardo.guerra@inpe.br)

Abstract: An Adaptive Object Model (AOM) is a common architectural style for systems in which classes, attributes, relationships and behaviors of applications are represented as metadata consumed at runtime. This allows them to be very flexible and changeable at runtime, enabling their modification by end users without source code modification. Nevertheless, this flexibility comes with a cost of a greater complexity when developing the system, and therefore one usually uses a bottom-up approach, adding flexibility only when it is needed. As a consequence, many AOM components are tied to the specific domain of a single application and this fact makes it difficult to develop and use generic and reusable AOM frameworks that properly handle specific requirements of the AOM architecture. This work presents an architectural model that aims to adapt domain-specific AOM core structures to a common core structure by identifying AOM roles played by each element through custom metadata configuration. By doing this, this model allows the integration of domain-specific AOM applications and AOM frameworks, making it feasible to develop reusable components for the AOM architecture. This model is evaluated by creating an AOM framework and a case study based on it, in which is performed a modularity and a performance analysis.

Keywords: framework, metadata, modularity, architecture, adaptive system, decoupling, Adaptive Object Model.
Categories: D.1.5, D.2.2, D.2.10, D.2.11, D.2.13

1 Introduction

Adaptive Object Model (AOM) is an architectural style that aims to provide the capacity of adaptation to a system domain model. That allows the introduction of new domain entities and the modification of existing ones at runtime, without changing the source code. The Illinois Department of Public Health (IDPH) Medical Domain Framework [Yoder et al. 2001; Yoder, Johnson 2002] is an example of a real system in production for more than 10 years that uses this approach. The reference cited presents only an initial subset of its design.

AOM architecture flexibility is achieved by representing classes, attributes, relationships, and operations as instances at runtime. The metadata used to represent the information of actual entities needed by the system, which can be changed by the application end users. This flexibility allows the domain to evolve as part of the business. However, this flexibility brings as a tradeoff a complexity in the

implementation of the system architecture, because it needs to handle common issues, such as persistence, presentation, and validation, for this flexible model.

The complexity in the implementation of such requirements would generate a smaller impact on the system development if it were possible to reuse existing components. However, since the flexibility is introduced in the model only where it is required, the AOM components are tied to their specific application domain [Ferreira et al. 2010a]. As a consequence, the components developed for them are hard to be reused in other applications that adopt the same architectural style.

This research project goal is to enable the reuse of existing components on AOM architecture, however allowing the application to define its own domain-specific model. That could potentially reduce the implementation time by reusing existing components, without losing the proximity of the model to the application domain.

This work presents an architectural model that adapts an AOM core structure coupled with a specific domain to a common AOM core structure by using metadata to identify the AOM roles played by classes, attributes and methods in the domain-specific AOM application. This solution externalizes the AOM core structure from the application domain and provides a common structure to be used by generic frameworks that implement AOM common requirements. The Esfinge AOM Role Mapper framework [Guerra 2012] was developed based on the proposed model. To evaluate this model, a modularity and performance analysis was performed on a case study application that used the developed framework. This paper is the extension of a previous paper published in SBCARS [Matsumoto and Guerra 2012], adding a further explanation about the proposed model, a performance evaluation and the support for hybrid models.

This paper is organized as follows: Section 2 gives an overview of AOMs; Section 3 presents the motivation for the creation of the architectural model presented in this work; Section 4 presents the architectural model proposed in this work; Section 5 presents the Esfinge AOM Role Mapper framework, which implements the model presented in Section 4; Section 6 introduces hybrid models and how they can be implemented with the framework; Section 7 presents a case study which assessed the impact of the proposed solution in system modularity and performance; Section 6 presents related works; and Section 9 presents the main conclusions and future work.

2 Adaptive Object Models

In a scenario in which business rules are constantly changing, implementing up-to-date software requirements has been a challenge. Currently, this kind of scenario has been very common and requirements usually end up changing faster than their implementations, resulting in systems that do not fulfill the customer needs and projects that have high rates of failure.

According to [Ferreira et al. 2010a], while software engineering methodologies, like Agile Software Development, try to increase the ability of adapting to changes, they consider each outcome of an iteration to be the last one, although it is not. Opposed to this approach, AOMs are developed to be incomplete by design [Garud et al. 2008].

In the AOM architectural style, classes, attributes, relationships and behaviors are represented using metadata [Yoder et al. 2001; Yoder, Johnson 2002], and

represented at runtime as instances. This allows the model to be changed at runtime and makes it possible to empower end-users to change the system according to their necessities, potentially reducing the time-to-market for modification in the domain model.

AOM architectures are usually made up of several smaller patterns, such as TYPE OBJECT, PROPERTY LIST, TYPE SQUARE, ACCOUNTABILITY, STRATEGY, RULE OBJECTS, COMPOSITE, BUILDER and INTERPRETER. Besides those, there are many other patterns that are used when creating an AOM application. These patterns form a pattern language for AOMs that is divided into six categories: Core, Process, GUI, Creational, Behavioral and Miscellaneous/Instrumental [Welicki et al. 2007a].

The Core category includes patterns that are present in basic implementations of AOMs and guides this architectural style. This work has focused on a subset of the Core category patterns composed by TYPE SQUARE (TYPE OBJECT and PROPERTIES) and ACCOUNTABILITY. These patterns are used for developing the structural part of an AOM core design and are described in the following sections.

2.1 Type Object

The TYPE OBJECT pattern [Johnson and Wolf 1997] is used in situations in which the number of subclasses that a class may need cannot be determined at development time. This pattern solves the situation by representing the subclasses that are unknown at development time as instances of a generic class that represents the object type.

Fig. 1 depicts the solution of the TYPE OBJECT. The unknown subclasses are represented as instances of the EntityType class. The Entity instances, which represent the actual instances of the system, refer to the EntityType instance that represents their class.

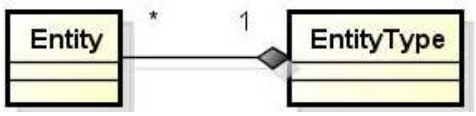

Figure 1: Type Object Structure.

2.2 Property

In situations in which instances of the same class can have different types of properties, to create an attribute to represent each of these properties in the class might not be the best solution. For instance, in a medical system one may create a class called Person to store information on patients, such as height, weight and blood type.

One solution would be to add an attribute to the Person class for each type of information that is necessary for the patient. However, if a hospital has different departments that need different kinds of information, one would probably need a great number of attributes in the Person class and just a few of them would effectively be used by an instance of this class (only those needed by the department in which the patient is being treated).

PROPERTY [Fowler 1996] solves this problem by representing the properties of an entity with a class and making this entity to have a collection of instances of this class. Applying the solution to the example, a Measurement class could be created to represent data from the patient. With this change, the attributes of the Person class could be replaced by one collection of Measurements, which would contain all and only the necessary measurements needed from one patient. Fig. 2 depicts the solution.

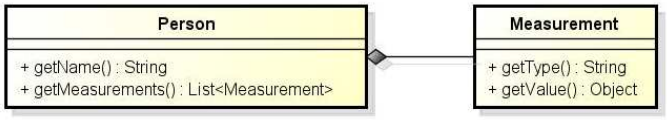

Figure 2: Property pattern applied to the example (adapted from [Fowler 1996]).

2.3 Type Square

In the AOM architectural style the TYPE OBJECT and PROPERTY patterns are usually used together, resulting in the TYPE SQUARE [Yoder et al., 2001]. In this pattern, the TYPE OBJECT is used twice – once for representing the entities and entity types of the system; and once for representing the properties and property types. Fig. 3 depicts the TYPE SQUARE structure.

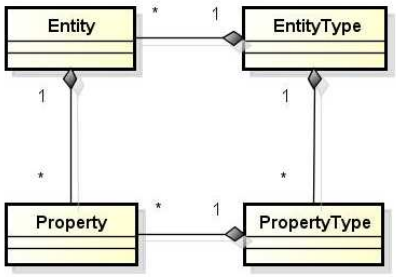

Figure 3: Type Square Structure (adapted from [Yoder et al., 2001]).

In this pattern, the EntityType and PropertyType classes represent the model and through their association it is possible to determine what kinds of properties are applicable to a given type of entity. The Entity and Property classes are related to the representation of the actual instances of the system. Each instance of Entity refers to an instance of EntityType that represents its type.

For each PropertyType in an entity's type, a Property is created to store the value of the property type in the entity. The class PropertyType defines the allowed properties for a given EntityType, and can also define some constraints such as its data type and allowed values.

With the TYPE SQUARE new types of entities with different types of properties can be created. Likewise, existing types of entities can be changed at runtime since modeling is done at instance level.

2.4 Relationship Representation

In an entity there are usually two kinds of properties: those that refer to primitive data types (attributes) and those that refer to relationships between entities (associations). In the AOM architectural style there are different ways to separate attributes from associations [Yoder et al., 2001]: (a) Create two lists of PROPERTY on the class, one for attributes and other for associations; (b) Make two subclasses of a Property class – Attribute and Association; (c) Check the type of the value of a Property object: a Property whose value is an Entity represents an association, while a Property whose value is a primitive data type is an attribute; (d) Use ACCOUNTABILITY [Fowler 1996] to represent the association.

While any of these options can be used for developing an AOM application, relationships are frequently represented by ACCOUNTABILITY in AOM core design diagrams.

ACCOUNTABILITY [Fowler 1996] allows the relationship between entities to be represented by an object (usually an instance of an Accountability class). Each Accountability object is associated to an AccountabilityType object, which represents the type of the relationship. Since the associations between entities are represented at the instance level, types of entity relationships can be created or modified at runtime, which makes this pattern suitable to the AOM architectural style.

2.5 Adaptive Object Model Core Design

The core design of an AOM system is depicted in Fig. 4. The diagram is divided in two parts – the operational level and the knowledge level [Fowler 1996].

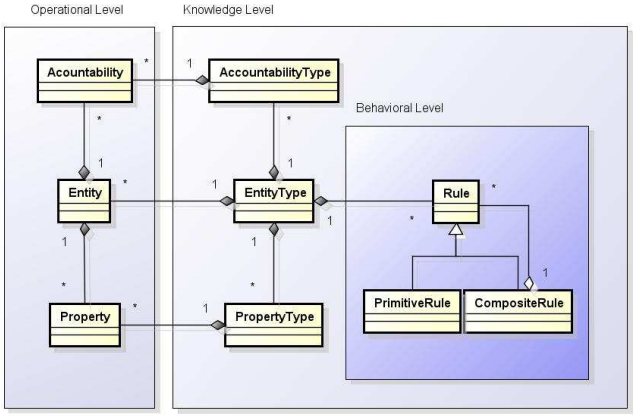

Figure 4: AOM Core Design (adapted from [Yoder et al., 2001]).

The instances of the classes in the operation level store the system's data and day-to-day events of the domain, while the instances of the classes in the knowledge level contain the representation of the system model. The behavioral level is responsible for handling business rules in the architectural style and usually uses STRATEGY [Gamma

et al. 1994] and RULE OBJECT [Arsanjani 2001]. The behavioral level was left out of the scope in the current stage of this work.

2.6 Other Concerns in AOM Applications

The flexibility provided by AOMs comes with a cost of a higher complexity when developing the application. Besides the fact that in AOMs metadata is used to represent the actual model of the system, developers also have to consider how to handle some implementation issues [Yoder and Johnson, 2002], such as:

(a) Persistence: not only should the actual data of an AOM be persisted, but also the representation of the model (described using metadata) should be stored in a database. The evolutionary nature of this model makes relational databases not the most appropriate type of storage. Another point to consider is how the system will be able to read information stored in the database and populate the AOM with the correct configuration of instances. Patterns like AOM BUILDER [11] should be considered when developing this issue.

(b) GUI: due to the dynamic nature of AOMs, user-interfaces have to be developed to be able to automatically adapt to changes in the model. In order to implement that, rendering patterns for AOMs [Welicki et al. 2007b] should be considered.

(c) Model Maintenance Tools: AOMs generally need tools and support GUIs to define and evolve the types in the system. These tools would be used for describing and maintaining the business rules of the application.

(d) **Version Control:** in order to support the evolution of the model in AOMs there is a need to implement a version control mechanism. Data of objects in the operational level must comply and be consistent with the model in the knowledge level. There is also a need to implement mechanisms to avoid the model to be broken due to partial updates.

Besides the issues presented above, there are many other points to be considered, such as security, instance validation, etc. All implementations, including business rules, related to the application domain must be based on metadata, because the system model is in the instance level and is not available at compile time.

3 Motivation

As mentioned in the previous section, there are some common concerns that should be handled when developing an AOM application. For a great number of these concerns there are patterns that help the development of the system. The solution presented by these patterns usually considers the core structure of AOMs (formed by the patterns TYPE OBJECT, TYPE SQUARE, PROPERTY and ACCOUNTABILITY) and could be implemented with a more generic AOM framework. However, since the core structure of AOM applications is usually coupled with the domain of the problem they solve, applications are not easily integrated with generic AOM frameworks.

In order to illustrate this issue, two systems that were modeled using AOM are considered in this example: the Illinois Department of Public Health (IDPH) Medical Domain Framework [Yoder et al. 2001; Yoder, Johnson 2002] and a banking system

for handling customer accounts [Riehle et al. 2000]. This example shows that although both systems share some common structures and needs, code cannot be reused among them because their core structures are coupled with their specific domains.

The IDPH Medical Domain Framework was developed in order to manage common information that was shared between applications used by the IDPH. This common information consists of observations made about people and relationships between people and organizations. Examples of these observations are blood pressure, cholesterol, eye color, height and weight.

In order to avoid the need for development and recompilation of the system whenever a business rule changed or a new type of observation was added, the application was developed using AOM. The resulting system model is depicted in Fig. 5. The design considers situations in which one observation is composed by other observations and also considers different types of observations (range values and discrete values).

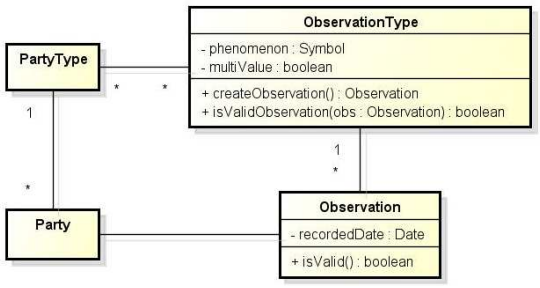

Figure 5: AOM Core Design (adapted from [Yoder et al., 2001]).

The example given in [Riehle et al. 2000] consists of a banking system for handling customer accounts. The fact that the number of types of accounts in the bank can increase significantly is taken into consideration and in order to avoid a subclass and attributes explosion the TYPE SQUARE pattern is used. The basic design for the system is shown in Fig. 6.

Notice the similarities between the structures used in the systems outlined above, such as the usage of the TYPE SQUARE pattern. Both systems present concerns like a persistence mechanism, a GUI, a version control for the object model and support tools for allowing end user development in the systems.

Although the systems share some common core patterns and have common needs, a framework developed for IDPH cannot be used for the banking system and vice versa, because each application is focused on solving the problems in their specific domains. As an example, a persistence framework developed for the IDPH system would be coupled to the medical domain and therefore it could not be used for handling persistence in the banking system.

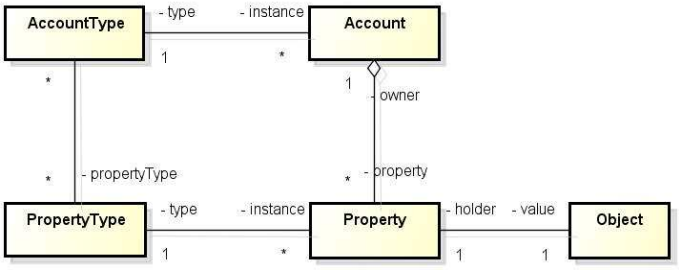

Figure 6: Basic design for the banking system [Riehle et al. 2000].

4 An Approach for Mapping AOM Models

In order to solve the integration problem presented in Section III, this work proposes an architectural model for a metadata-based framework that adapts the domain-specific AOM core structures to a common AOM structure. For the sake of clarity, this framework is referred as the *integration framework* and any client of this framework (i.e. generic AOM frameworks and client applications) is referred as *client*.

In the solution proposed by this work, the integration framework provides a common AOM core structure that can be referred by generic AOM frameworks. The framework also provides classes that adapt the domain-specific AOM core structures to this common AOM structure. Since these classes implement the ADAPTER pattern [Gamma et al. 1994], they are referred as the *adapter classes* in this work.

In order to be able to adapt domain-specific AOM core structures, the integration framework only needs to identify at runtime the roles that classes, methods and attributes of domain-specific classes play in the AOM architecture. Examples of these roles are Entity, Entity Type, Property and Property Type. This identification is accomplished by the use of metadata resources. The inference of the roles can be possible in some scenarios, but since different kinds of implementation are possible, that is not always the case.

Fig. 7 shows the representation of the solution applied to the IDPH and Banking System examples given in Section III. In the figure, the metadata used for identifying the AOM roles are Java annotations. As depicted, the domain-specific classes are marked with specific annotations that are consumed by the adapter classes, which implement the interfaces that define the common AOM core structure. When a method is called in an adapter object, it is able to know which exact method to call in the adapted object due to the AOM role annotations. The generic AOM frameworks only need to refer to the interfaces of the common AOM core structure provided by the integration framework.

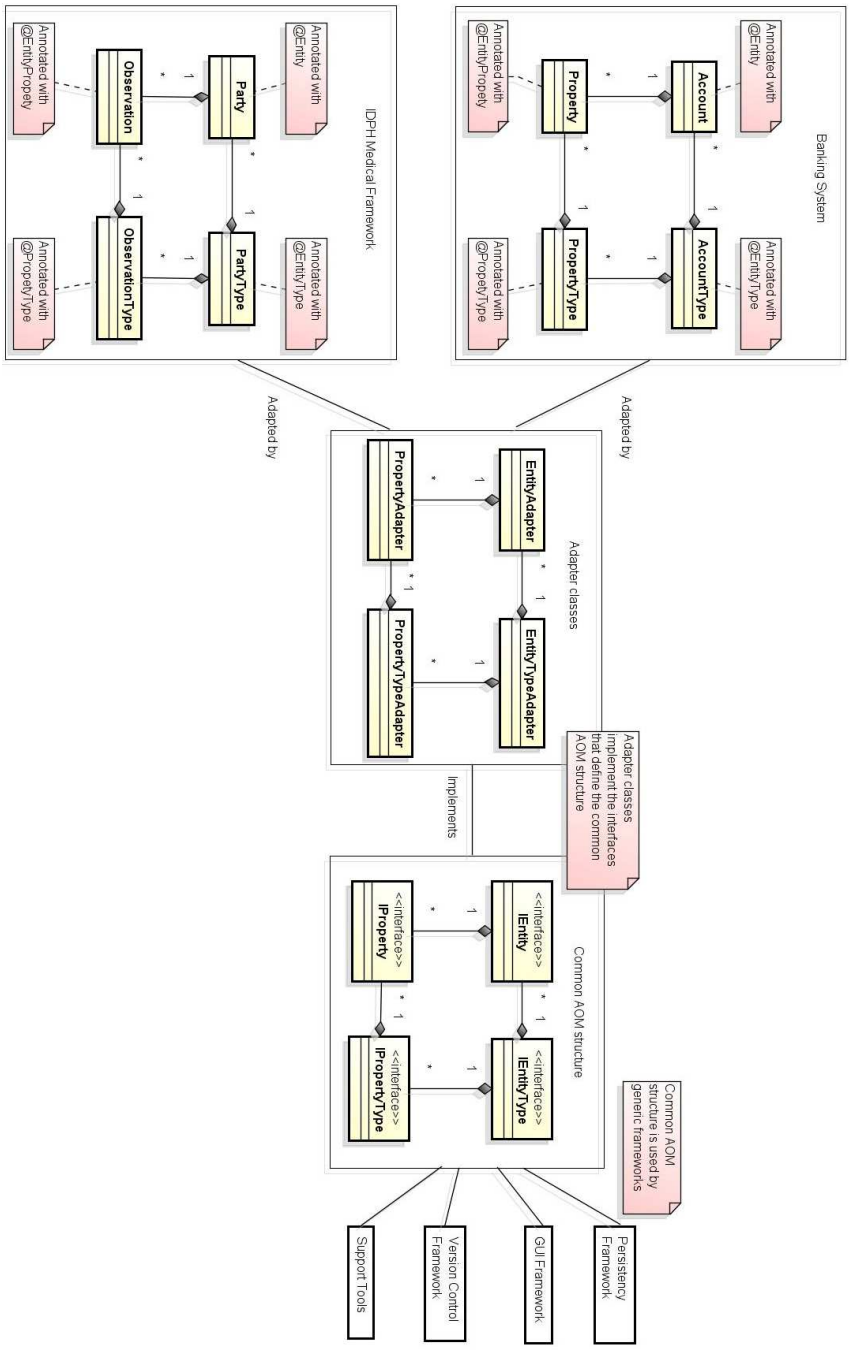

Figure 7: Representation of the solution for the examples given in Section 3.

With this approach, domain-specific applications can be integrated with generic AOM frameworks only by identifying the AOM roles played by its classes, attributes and methods in the AOM core structure, using the metadata provided by the integration framework. All the responsibility for the integration is left outside the domain-specific applications and the generic AOM frameworks. Notice that the only change to the domain-specific applications is the inclusion of metadata to identify the AOM roles. Therefore, the applications can continue using their domain-specific AOM core structure without any change to the solution's architecture.

The use of metadata for identifying the AOM roles allows the integration framework and the domain-specific applications to be completely decoupled if external metadata, like XML, is used or loosely coupled if metadata such as annotations and custom attributes are used. Besides, notice that the domain-specific AOM applications only depend on the metadata provided by the integration framework for identifying the AOM roles of their classes, methods and attributes. Additionally, generic AOM frameworks only have to refer to the common AOM core structure provided by the integration framework.

4.1 Core Components

The main components for developing an integration framework are the following: **(a) Metadata Handler** is responsible to retrieve metadata from the application classes. It implements metadata reading patterns [Guerra et al. 2013a] to decouple metadata handling operations from the rest of the framework; **(b) AOM Core API** includes a set of interfaces that represent the common AOM core structure provided by the framework; **(c) AOM Core Implementations** contains implementations of the interfaces defined by the AOM Core API component. There are two types of implementations in this component: a basic and general implementation of the AOM core structure and an implementation that adapts domain-specific AOM core structures using the Metadata Handler component; **(d) Model Manager** is responsible to instantiate the model and manage the instances of the AOM Core API created by the framework.

Fig. 8 depicts the relationship between the main components in the integration framework. In this representation, the component Client represents the clients of the proposed framework.

The client uses the Model Manager component to perform operations over the model, such as loading and saving elements of the architecture. The client and the Model Manager components are able to perform operations on the AOM Core Implementation objects through the AOM Core API component interfaces. When an operation is performed over an adapter object of the AOM Core Implementation component, this object gets information on the domain-specific AOM core structure metadata by using the Metadata Handler component. This way, the adapter object is able to perform the corresponding operation on the adapted domain-specific AOM application object.

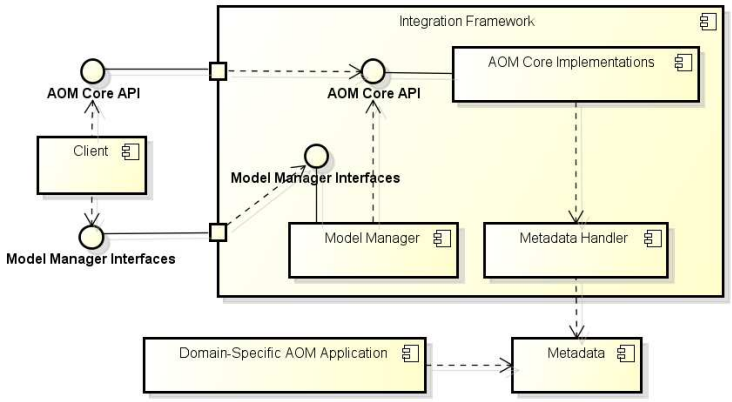

Figure 8: Relationship between the components in the integration framework.

4.2 Rationale

As shown in the previous sections, the architectural model proposed in this work is able to integrate domain-specific AOM applications and generic AOM frameworks by adapting the domain-specific AOM core structures to a common AOM core structure defined by the AOM Core API component.

The only information needed for adapting the domain-specific AOM core structure is the AOM roles played by the elements in the structure. These roles are identified through metadata that is consumed at runtime by the integration framework, using the Metadata Handler component. Notice that there is only a weak dependency between the domain-specific AOM applications and the integration framework. The applications only have to mark their core structures with the metadata provided by the framework. If external metadata is supported, there is no need of changes in the domain-specific applications' code. Otherwise, only minor changes, such as inserting an annotation to the code, is required in order to make the application be adaptable by the integration framework.

In the generic AOM frameworks perspective, no knowledge related to domain-specific applications is needed. These frameworks simply make use of the components provided by the integration framework (AOM Core API and Model Manager) in order to be applicable to any domain-specific application that is configured to be adaptable by the integration framework.

As a consequence, with this architectural model, generic AOM frameworks and domain-specific AOM applications can be integrated, even though being completely decoupled. Due to this possibility, generic AOM frameworks can be developed without being tied to any specific domain and can be applied to different AOM applications, allowing reuse of code and design.

5 Esfinge AOM Role Mapper Framework

The Esfinge AOM Role Mapper integration framework [Guerra 2012] was developed in order to evaluate the proposed model. Besides the functionality of adapting existing domain-specific AOM core structures to a common structure, the framework also allows the creation of an AOM application from scratch. Despite this framework is implemented in Java; the model can be considered language-independent and could be built to other platforms. The following sections give an overview of the framework components.

This framework was created in the context of the Esfinge project (http://esfinge.sf.net), which is an open source project that comprises several metadata-based frameworks for different domains. Examples of other frameworks developed in this project were Esfinge QueryBuilder [Guerra 2014] for generating database queries based on method signatures, Esfinge Guardian [Silva et al. 2013] for access control and Esfinge SystemGlue [Guerra et al. 2013b] for application integration. The development of these projects is also used in a research focusing on identifying models, patterns and best practices for metadata-based frameworks.

5.1 Metadata Handler Component

The Metadata Handler component implements some of the patterns of the pattern language presented in [Guerra et al. 2013a] and can be divided in the following parts:

(a) Descriptors: implements the METADATA CONTAINER pattern. Each role in an AOM architecture is represented by one descriptor which contains references to get/set/add/remove Method objects for each relevant field

(b) Metadata Readers: implements the METADATA READER STRATEGY pattern. Currently, the framework only supports annotations for determining the AOM roles of elements in domain-specific applications, but since the METADATA READER STRATEGY pattern was implemented, it supports extensions related to the support of other types of metadata.

(c) Metadata Repository: implements the METADATA REPOSITORY pattern, providing an in-memory cache of the metadata already retrieved.

(d) Annotations: contains the Java annotations that allow the identification of the AOM roles of the elements in the domain-specific AOM applications. The names of some annotations created for the framework are similar to some JPA's annotations, but they are completely unrelated. The annotations in Esfinge AOM Role Mapper are used to map elements of domain-specific AOM applications to the generic AOM core structure.

5.2 General AOM Model Implementation and API

The common AOM core structure provided by the framework consists of the following interfaces: IEntityType, IEntity, IPropertyType and IProperty. These interfaces are implemented by classes in two different packages – one that contains implementations related to the adaptation of domain-specific AOM core structures to the common core structure provided by the framework; and another that contains generic AOM classes that can be used for creating a new AOM application using the

framework. The framework provides factory classes that are able to decide what class to instantiate according to parameters passed to the creation methods.

Although these two types of core structure implementations are available, this paper focuses on the implementation of the ADAPTER classes, which differentiates this approach from the other existing frameworks. The ADAPTER core structure is composed by five classes: AdapterEntityType, AdapterEntity, AdapterPropertyType, AdapterProperty and AdapterFixedProperty. Each of these classes contains an attribute for storing the domain-specific AOM application object that they adapt.

By considering the example of a domain-specific class annotated with the metadata provided by the Esfinge AOM Role Mapper framework depicted in Fig. 9, when an AdapterEntity object is created to adapt an Account object, the Metadata Handler component is used to get the descriptor for the Account class. This descriptor will contain the Method objects for getting and setting the account's account type, among other data. Using the information provided by the descriptor, the AdapterEntity object is able to invoke methods over the Account object.

```
@Entity
public class Account {
   @EntityType private AccountType accountType;
   public AccountType getAccountType() {
      return accountType;
   }
   public void setAccountType(AccountType accountType) {
      this.accountType = accountType;
   }
}
```

Figure 9: Example of a domain-specific application class with annotations.

Fig. 10 shows an example of how the getEntityType() method is adapted by an AdapterEntity object. When the Client calls the getEntityType() method, the AdapterEntity object obtains the Method instance (from Reflection API) that gets the Entity Type of the adapted entity from its metadata descriptor. Then, it invokes this method using reflection and obtains the domain-specific Entity Type instance for the adapted Entity.

The getEntityType() method must return an IEntityType object and, therefore, it calls the getAdapter() static factory method of the AdapterEntityType class, passing the domain-specific Entity Type object as a parameter.

This method queries an internal map in the AdapterEntityType class, which relates a domain-specific object to the AdapterEntityType instance that adapts this object. The use of this map avoids the creation of more than one object to adapt the same domain-specific object – if an AdapterEntityType object was already created for adapting a determined domain-specific Entity Type object, the getAdapter() method only returns the previously created object; otherwise, it creates a new instance of AdapterEntityType, puts it into the map and returns it. Finally, the object returned by the getAdapter() method is returned to the Client.

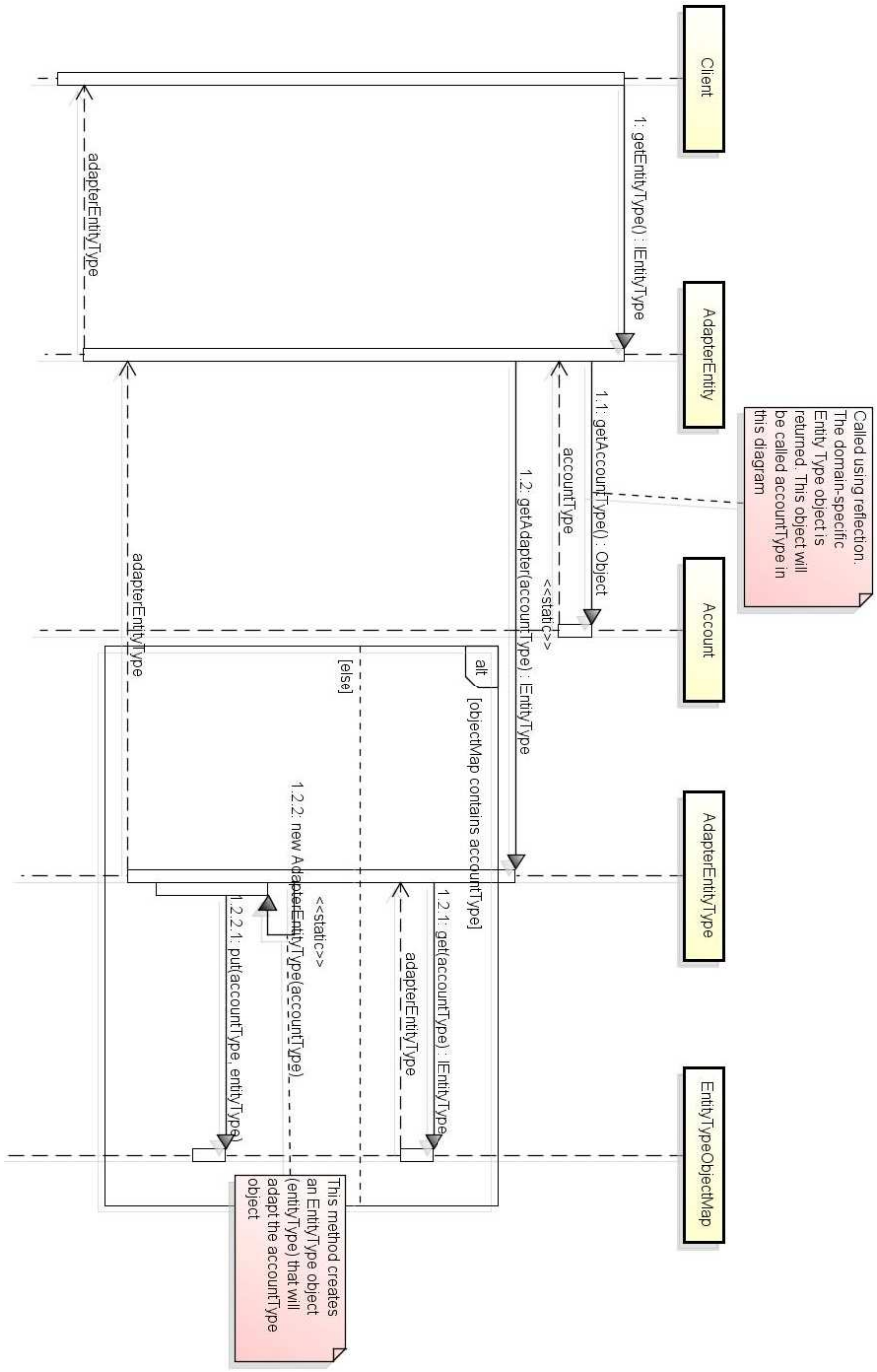

Figure 10: Sequence diagram showing how domain-specific objects are adapted.

The adaptation process described above is the overall solution used for the adapter classes of the Core Implementations component (i.e. AdapterEntityType, AdapterEntity, AdapterPropertyType, AdapterProperty).

5.3 Mapping Annotations

Domain-specific AOM applications need to have the elements in their core structures marked with the metadata provided by the Esfinge AOM Role Mapper framework. Currently, the only metadata type supported by the framework is Java annotations, however it provides a hotspot on the metadata reader making possible to extend it to support other approaches for metadata definition, such as code conventions, XML files or database. The following list describes the annotations defined by the framework and the classification of the annotation as "type", "field" or "method" refers to each kind of element it can be associated:

- ***@EntityType***: Type and field annotation, which allows the identification of classes that play the Entity Type role in the AOM architecture and also the field in Entity classes which points to the corresponding Entity Type.
- ***@Entity***: Type annotation, which allows the identification of classes that play the Entity role in the AOM architecture.
- ***@PropertyType***: Type and field annotation, which allows the identification of classes that play the Property Type role in the AOM architecture. It also allows the identification of the field in Property classes which points to the corresponding Property Type and the field in the Entity Type classes which points to the Property Types defined for the Entity Type.
- ***@EntityProperties***: Type and field annotation, which allows the identification of classes that play the Property role in the AOM architecture and also the field in Entity classes which points to the Properties of the Entity.
- ***@EntityProperty***: Field annotation, which allows the identification of fixed properties in Entity classes.
- ***@Name***: Field annotation, which allows the identification of the field that determines the name of an Entity Type or a Property Type.
- ***@PropertyTypeType***: Field annotation, which allows the identification of the field that determines the type of a Property Type.
- ***@PropertyValue***: Field annotation, which allows the identification of the field that determines the value of a Property.
- ***@CreateEntityMethod***: Method annotation, which allows the identification of the method in the Entity Type classes that handles the creation of an Entity object. If a method marked with this annotation is not present in an Entity Type class, it is not possible to create an entity using the createNewEntity method of the AdapterEntityType class, because the framework will not be able to know which Entity class should be created.

5.4 Model Manager Component

The Model Manager component's responsibility is to orchestrate the instances created by the Esfinge AOM Role Mapper framework. The main class of this component is the ModelManager, whose instance is unique. All the operations involving the

manipulation of the model, including model persistence, loading and querying, should be done through this class. For accessing the database, the ModelManager makes use of the IModelRetriever interface, which can be implemented by persistence frameworks.

In order to get the instance of IModelRetriever to be used, the ModelManager class uses the Service Locator functionality that is available in the standard Java API. The advantage of using Service Locator is that the implementation of the IModelRetriever interface becomes totally decoupled from the framework, allowing great flexibility. It also allows services to be dynamically changed using decentralized configuration.

One of the main responsibilities of the ModelManager class is to guarantee that a logical element is not instantiated twice in the framework. In order to control that the ModelManager contains two Map objects – one for storing the loaded Entities by their IDs and one for storing the loaded Entity Types by their IDs. Whenever a method that loads an Entity or an Entity Type is called, the ModelManager checks whether the ID of the instance to be loaded is already found in the corresponding map. If so, it returns the previously loaded object. Otherwise, it calls the IModelRetriever object for loading the object into the memory and saves it into the map.

6 Hybrid Models

In real applications that use AOM as the architectural style, usually the flexible domain model is implemented only in entities where it is a requirement. Other entities often follow a static model adopted by the target programming language, containing fixed attributes and accessor methods. Even in AOM entity types, for instance, there can be static properties that should be present in all entities. The present paper defines a model that contains static classes, AOM entities and other intermediate solutions as ***Hybrid Model***.

The implementation of components for domain-specific hybrid models is straightforward because they can have specific code to handle the static properties and classes. Indeed, the existence of these exceptions to the "pure" AOM model is a core factor that disables its component reuse in other contexts.

In order to support hybrid models, fixed properties are adapted by Esfinge AOM Role Mapper framework using AdapterFixedProperty objects, which hold an object that plays the Entity role in the domain-specific application. This approach is different from the AdapterProperty objects that hold a corresponding Property object in the domain-specific application. The get/set methods for the adapted attribute are invoked in order to obtain and set the value for the AdapterFixedProperty object.

Another characteristic of the AdapterFixedProperty objects is that the property type objects they refer to are instances of the GenericPropertyType class, which is the non-adapter implementation of the IPropertyType interface in the framework. Since fixed properties are standard attributes in the domain-specific Entity class, there is no corresponding Property Type object to be adapted by the framework. Therefore, a GenericPropertyType object must be created in order to represent the type of the property, with the name of the field and the type of the field.

Since the AdapterFixedProperty class implements the IProperty interface, the differences between this class and the AdapterProperty class are internal to the

framework. As a result, for the frameworks that handle the properties of an entity type, it is transparent if the property is defined statically or dynamically in the domain-specific model. The main difference is that the framework does not allow fixed properties to be added or removed.

In Fig. 11, an example of a domain-specific class with a fixed property is shown. Notice that when an instance of AdapterEntityType for adapting an AccountType object is created, it must have a reference to the GenericPropertyType object that is related to the accountNumber fixed field.

```
@Entity
public class Account {
   @EntityType private AccountType accountType;
   @EntityProperty private int accountNumber;
   public int getAccountNumber() {
      return accountNumber;
   }
   public void setAccountNumber(int accountNumber){
      this.accountNumber = accountNumber;
   }
}
```

Figure 11: Example of a domain-specific class with a fixed property.

In Fig. 11, an example of a domain-specific class with a fixed property is shown. Notice that when an instance of AdapterEntityType for adapting an AccountType object is created, it must have a reference to the GenericPropertyType object that is related to the accountNumber fixed field.

The class with only fixed properties can be considered the worst case for AOM adaptation, because it does not provide any flexibility to change the domain model. However, there were other intermediary scenarios, where there is some flexibility, but it is not a complete TYPE SQUARE implementation yet. Figure 12 presents some possible paths for evolving an AOM application from a static model to the complete TYPE SQUARE implementation.

Esfinge AOM Role Mapper considered that in a hybrid model several stages of an AOM implementation could be implemented, which includes all the steps presented in the evolution path. Tests were performed considering every step on Figure 12, to make sure that in a domain model, several stages of flexibility could be present on different entities. This support also allows the refactoring of the application towards an AOM allowing the framework to understand the model in every step of this evolution.

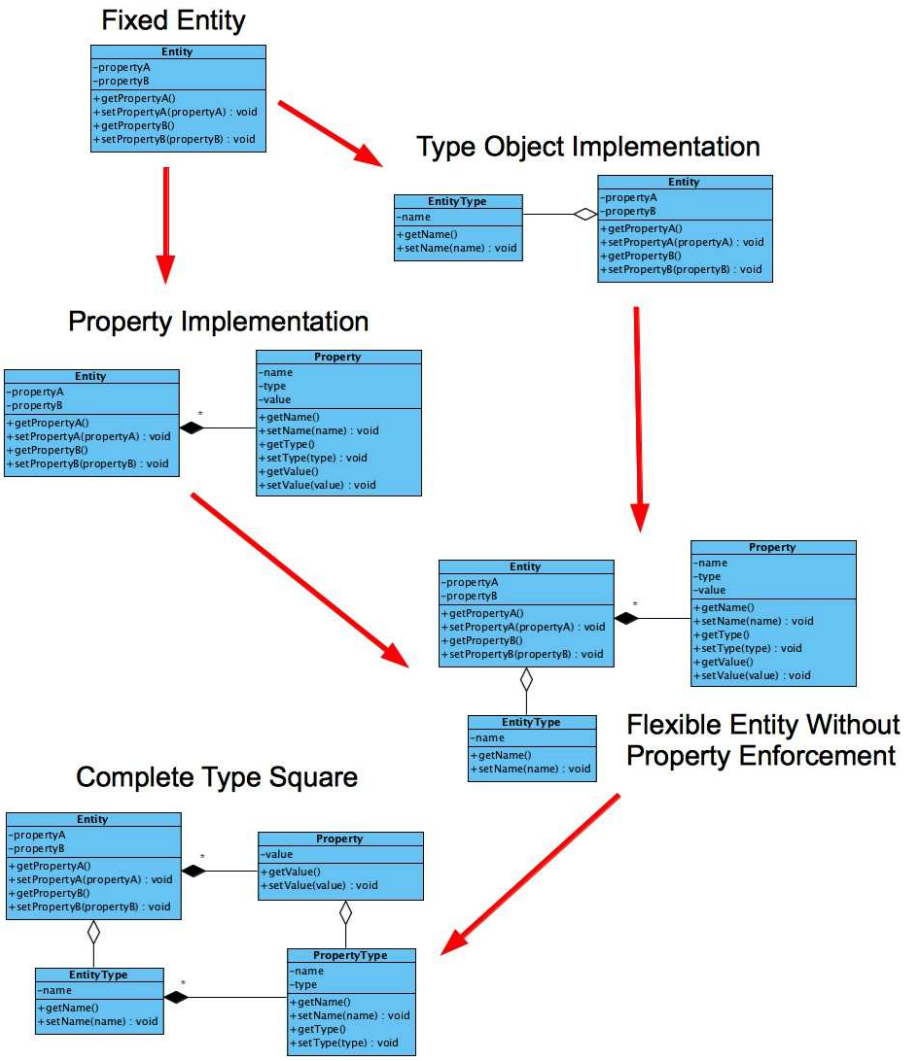

Figure 12: AOM evolution path supported by Esfinge AOM Role Mapper.

7 Evaluation

The goal of this section is to describe the analysis performed to evaluate the approach proposed regarding the performance overhead and the decoupling provided. The next subsections present the evaluation goal, its approach, the software artifacts developed, the analysis performed and its threats to validity.

7.1 Evaluation Goal

The evaluation has two different goals: (a) to verify if by following the proposed approach it is possible to have AOM frameworks decoupled from the AOM domain-specific models; (b) to measure the performance overhead introduced by the framework AOM adapters.

The decoupling verified by the first goal is a requirement to enable the reuse of AOM frameworks in different models. Despite the fact that the achieved reuse is not measured, the decoupling of two components of an application suggests that one can be changed without affecting the other, allowing them to be used on other contexts. The second goal aims to measure the performance price paid for the flexibility provided by the proposed approach usage.

7.2 Evaluation Approach

In this evaluation two distinct domain-specific AOM models taken from the literature were implemented. Additionally, two distinct frameworks to handle different concerns were also created. Based on these implementations and using the Esfinge AOM Role Mapper, applications were instantiated. As a requirement, the applications should be able to create new types and new instances based on the AOM model.

After that, considering the software artifacts used to create the applications, a modularity analysis was performed. This analysis was based on a Dependency Structure Matrix (DSM) [Steward 1982; Yassine 2004], which shows the package dependencies between all the jar files involved. The evaluation is considered successful if the application works as expected and if the modularity analysis does not find any dependence between the frameworks and the AOM models.

By using the two models developed as reference, a performance measurement was performed considering the original classes and the classes adapted by the framework classes to the AOM Core API. The overhead found reveal the performance price for using the proposed approach.

7.3 Software Artifacts Developed

All the artifacts used in the evaluation are available as free software on Esfinge AOM Role Mapper repository (https://github.com/EsfingeFramework/aomrolemapper). Considering the evaluation approach described in the previous section, the following describe the frameworks and the AOM structures implemented:

- **Domain-Specific AOM Structures:** There were developed two domain-specific AOM core structures for the case study: one for a banking system and one for a medical system. It is important to state that these structures are based on examples referenced by other works [13, 1]. The domains of the systems are similar to the examples shown in Section III. Both structures were mapped by using the Esfinge AOM Role Mapper framework annotations.
- **Persistence Framework:** For validating the concept presented in this work, a persistence framework that implements the IModelRetriever interface was developed. This framework is called AOM Mongo Persistence framework. The database used for implementing the persistence framework was MongoDB [Membrey et al. 2010], which is a document-oriented storage and is more suited for dealing with the dynamic nature of AOMs than SQL databases.

- **Console-based User Interface:** The second framework developed for this evaluation is a simple console-based interface which shows menus for: loading the model into memory; saving the model; adding / removing / changing Entity Types; adding / removing / changing Entities; and showing Entities and Entity Types. It can be considered a framework because it contains hotspots to be adapted to any mapped AOM model, such as a different AOM implementations and different entity structures.

7.4 Executable Application

To execute the application, the Console-based User Interface framework was configured to work with both banking and medical system without the need to change any code. Additionally, the AOM Mongo Persistence framework was used as the persistence framework.

The list below shows the jar files that were used in the analysis: (a) aomrolemapper.jar: Contains the Esfinge AOM Role Mapper framework; (b) aompersistence.jar: Contains the AOM MongoDB Persistence framework; (c) bankingexample.jar: Contains the domain-specific banking AOM model; (d) medicalexample.jar: Contains the domain-specific medical AOM model; (e) aomtest.jar: Contains the Console-based User Interface framework.

When running the client application, the aomrolemapper.jar and aompersistence.jar must be included in the classpath. If the banking system is the one that needs to be adapted, the client application only has to include the bankingexample.jar file into the classpath. Similarly, if the medical system is the one to be adapted, the client application only has to insert the medicalexample.jar file into the classpath. It is also possible to adapt both systems simultaneously and use the client application without any changes to the code.

7.5 Modularity Analysis

In order to analyse the modularity of the AOM models and frameworks developed for the evaluation, a Dependency Structure Matrix [Yassine 2004] that shows the package dependencies in all the jar files involved was generated using the Lattix tool and it is presented in Fig. 13. The modules involved can be easily identified through the different colors. The 'X' character indicates that the module represented in the line depends on the module represented by the column. Any reference from one module to another (e.g. invocation of a method or the use of an annotation) is considered a dependency.

Notice that the domain-specific applications, represented by 21 and 22, only depend on the annotations package of the Esfinge AOM Role Mapper framework, represented by 13. This package only contains the definitions of the annotations provided by the framework.

Analyzing the dependency of the AOM Mongo Persistence framework package (1), it is possible to observe that the framework depends on the api (4) and exceptions (5) packages of the Esfinge AOM Role Mapper framework. The first package contains the IModelRetriever interface, which is implemented by the persistence framework; and the Model Manager and AOM Core API component interfaces. The

second package contains the exceptions thrown by the Esfinge AOM Role Mapper framework.

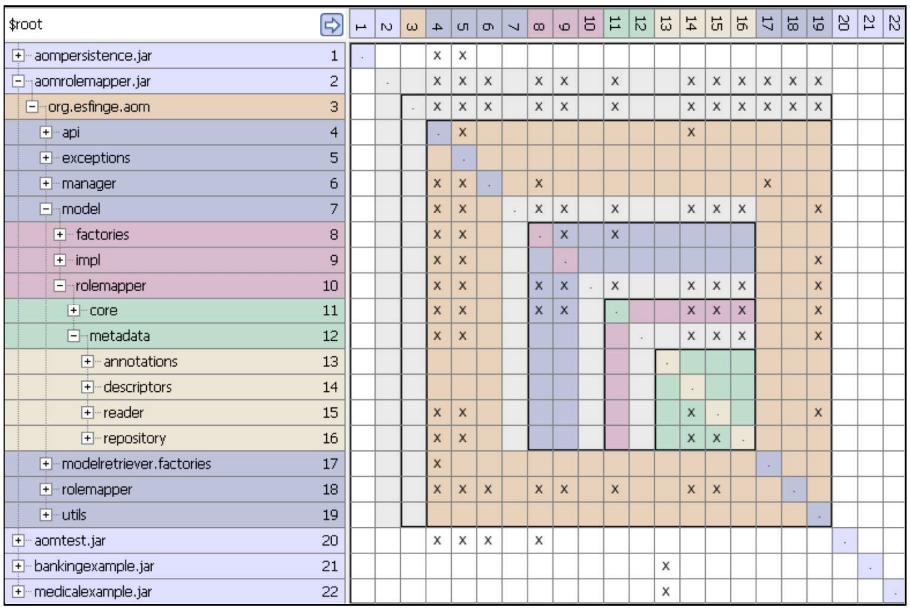

Figure 13: DSM generated from the result application.

Notice that the domain-specific applications, represented by 21 and 22, only depend on the annotations package of the Esfinge AOM Role Mapper framework, represented by 13. This package only contains the definitions of the annotations provided by the framework.

Analyzing the dependency of the AOM Mongo Persistence framework package (1), it is possible to observe that the framework depends on the api (4) and exceptions (5) packages of the Esfinge AOM Role Mapper framework. The first package contains the IModelRetriever interface, which is implemented by the persistence framework; and the Model Manager and AOM Core API component interfaces. The second package contains the exceptions thrown by the Esfinge AOM Role Mapper framework.

Notice that the persistence framework only depends on information exposed by the interfaces of Esfinge AOM Role Mapper framework. It does not depend on the specific implementation of this framework and nor on any information related to the domain-specific applications. The fact that the persistence framework only depends on the Esfinge AOM Role Mapper framework makes it applicable to any application that can be adapted by the Esfinge AOM Role Mapper framework.

Similarly, the console-based user interface (20) dependency analysis shows that it only depends on information exposed by the Esfinge AOM Role Mapper framework. It depends on the api (4), exceptions (5), manager (6), and model.factories (8) packages of the framework. The first two packages contents were previously

explained. The third package contains the ModelManager class and the forth one contains the factory classes for the common AOM core structure classes.

7.6 Performance Overhead Analysis

To evaluate the overhead brought with the AOM adapters, several tests were performed measuring the time necessary to execute actions, with and without the AOM adapters. The bank account domain-specific AOM model was used for these tests. The time measurement was performed in nanoseconds using the virtual machine method System.nanoseconds(). The time was measured executing the test a 1000 times and dividing the final time by 1000. Each test was performed 12 times and the value registered in a spreadsheet. Table 1 present the average value from these measurements.

Table 1: Time measured from the tests on the AOM model.

Test performed	**Time without adapter (ns)**	**Time with adapter (ns)**
Entity type creation	12227	409469
Property type creation	5991	24166
Entity creation of an entity type with 1000 properties	154519	1250243026
Setting a property in an entity type with 1000 properties	37409	1513474

Due to the small time for the creation of a domain-specific AOM model, the performance overhead is large in the scenario. However, absolutely, the numbers are small, in the order of nanoseconds. On the one hand, considering information systems that deal with database access and network communication, these values can be considered insignificant. On the other hand, this overhead can be significant if an algorithm repeats it several times.

The largest overhead was measured in the creation of an entity type, because a wrapper should also be created for all 1000 properties. It is important to highlight that it is not usual to have such high number of entity attributes. However, in future versions a lazy loading mechanism can be used to workaround this initial overhead.

7.7 Threats to Validity

This section presents some threats to validity that can compromise the results of this evaluation. The issues described here were considered in the analysis of the results and will be considered and addressed in further works.

The first issue that should be considered is the small size of the AOM models used. To avoid being tendentious for building the models, the authors choose to use existing models described by different papers in the literature, namely the Illinois Department of Public Health (IDPH) Medical Domain Framework [Yoder et al. 2001; Yoder, Johnson 2002] and a banking system for handling customer accounts [Riehle et al. 2000]. The models, as presented in these papers, are small and may not reflect the needs of realist software. However, they represent only the initial AOM

architecture of such systems, which surely have more complex requirements and solutions. Additionally, since the frameworks developed perform usual tasks, such as user interface and persistence; the application functionality does not contain complex scenarios in which the proposed model may not be enough for a decoupled implementation. A more complete evaluation could be done if medium or large-scale applications were used.

Another threat to validity is related to the measurement strategy. The DSM showed that it is possible to decouple the frameworks from the domain-specific AOM model, providing evidence that it is possible to reuse the framework in different contexts. However, no measurement was performed to evaluate the reuse itself, to access potential impacts on quality and team productivity.

Finally, since the Esfinge AOM Role Mapper framework only implements mapping for the basic AOM patterns, other elements that include the behavioral level were not included on the model. These new model elements may bring different consequences the application. This kind of model was considered out of scope and the conclusions should be considered restricted to the implementation of the core AOM patterns.

7.8 Evaluation Conclusions

In this evaluation, it was possible to verify that no code changes were needed in order to make the frameworks work with both domain-specific AOM models. The only actions needed were to change configuration files and to put the proper domain-specific application jar into the classpath.

The metrics depicted by the DSMs shows that the domain-specific models only depend on the annotations defined by the Esfinge AOM Role Mapper framework, which means that the domain-specific applications can still be used without the generic AOM frameworks and applications. The DSM also shows that the persistence framework and the console-based user interface only depend on information that is externalized by the Esfinge AOM Role Mapper framework, which means that they can be reused among different domain-specific AOM core structures.

These results show that the proposed architectural model accomplished its goal to decouple the core structure of AOM applications from their specific domains, providing a way to allow reuse of design and code of generic AOM frameworks and applications among different domain-specific applications. The performance measurements showed that the performance overhead is usually acceptable for information systems requirements. However, these conclusions are based on simple models that only implement the core AOM patterns, and further evaluations should be performed on more complex models.

8 Related Work

Dynamic languages are easily extensible and allow the addition of new members in a class, such as methods and attributes. While this functionality is straightforward to be used programmatically, the management of application entities is not ready to be performed by users such as proposed by the AOM architecture. Because of that, even in dynamic languages, such as Ruby and Javascript [Bhati 2009], an AOM structure

can be used to represent a dynamic domain model. Because of that, this section focuses on works that put effort on enabling reuse of AOM implementations.

In [Yoder and Johnson, 2002], many examples of systems that use the AOM architectural style are presented. While these systems aim at solving specific issues in specific domains, other frameworks, such as Oghma [Ferreira et al. 2009; Ferreira 2010], ModelTalk [Hen-Tov et al. 2009] and its descendant, Ink [Acherkan et al. 2011] aim at providing generic AOM frameworks for easing the creation of adaptive systems, mainly through the use of a Domain-Specific Language (DSL).

Oghma is an AOM-based framework written in C#, which aims to address several issues found when building AOM systems, namely: integrity, runtime co-evolution, persistence, user-interface generation, communication and concurrency [Ferreira et al. 2009]. The modules that handle each of these concerns reference the AOM core structure of the framework, which was developed to be self-compliant by using the EVERYTHING IS A THING pattern [Ferreira et al. 2010b].

Oghma allows a client program to instantiate a model for its domain by simply calling the constructor of the MetaModel class of the framework, passing as argument an XML model configuration file that contains the Entities descriptions. After this model is created, the aforementioned AOM requirements implemented by the framework are readily available.

ModelTalk and Ink are AOM frameworks that rely on a DSL interpreter to add adaptability to the model. At runtime, instances of DSL classes are instantiated and used as meta-objects for their corresponding Java instances through a technique called model-driven dependency injection [Hen-Tov et al. 2009]. Developers are able to change the model by editing the ModelTalk/Ink configuration in an Eclipse IDE plug-in specially developed to handle the framework DSL. When changes in the model are saved, the plug-in automatically invokes the framework's DSL analyzer, performing incremental cross-system validation similar to background compilation in Java.

What the Esfinge AOM Role Mapper framework presented in this work has in common with Oghma and ModelTalk/Ink is the fact that it is an AOM framework not tied to any specific domain and is intended to ease the development of AOM-based systems. However, instead of considering that the entire infrastructure for building AOM systems must be inside the framework, the Esfinge AOM Role Mapper provides a standard AOM core structure that can be used by different AOM related frameworks, such as persistence, GUI and version control frameworks.

Beside this fact, the Esfinge AOM Role Mapper framework is able to adapt the core structure of domain-specific AOM applications to the common structure it provides. As a consequence, this framework can be used for integrating generic AOM frameworks to existing domain-specific AOM applications. The Oghma and ModelTalk/Ink frameworks do not provide this functionality, which requires the applications to be created from scratch, coupling the AOM model to the framework.

Finally, even though the Esfinge AOM Role Mapper adapts the domain-specific core structures, these structures remain logically unchanged and can still be used in the system. This means that behavior can still be added through the application code, which brings simplicity to the system. But with Oghma and ModelTalk/Ink, the development of the system is limited to the expressiveness provided by those frameworks.

9 Conclusions

The Adaptive Object Model is an architectural style that provides great flexibility by representing classes, attributes, methods and relationships as metadata. The tradeoff of this architectural style is the higher complexity when implementing AOM systems. Therefore, AOM application developers tend to use bottom-up approaches, adding flexibility only where it is necessary. As a consequence, there are many AOM systems that are tied to the specific domain for which they were developed and this makes it difficult to create generic AOM frameworks that can be applied to any AOM application.

This work presented an architectural model to solve this issue by adapting the domain-specific AOM core structures to a common core structure by using metadata to identify AOM roles of elements in the domain-specific application. The code and design of generic AOM frameworks that use the common core structure can be reused by different AOM applications, even though they are tied to different domains. This work also presented the AOM Role Mapper framework, which implements the proposed model in Java and uses annotations as metadata. Although the proposed solution API seems similar to other mapping frameworks, such as ORM, the internal solution is very different from these frameworks since it has to cope with two models that can be dynamically changed.

The modularity analysis made over the case study in this work showed that the domain-specific AOM applications have a weak dependency on the Esfinge AOM Role Mapper framework. The analysis also showed that the AOM generic framework and the Client application created for the case study only depended on the Model Manager component and the common AOM core structure provided by the Esfinge AOM Role Mapper framework. No information related to the specific implementation of the framework or the domain-specific applications was needed by the generic AOM framework and the Client application.

Although it was possible to show that the proposed architectural model solved problems related to the integration of AOM generic frameworks and domain-specific AOM applications, there is still a great research field in this area. This work focused on the creation of the initial version of this integration framework, only supporting the adaptation of a basic AOM core structure and its variations. In order to have a framework that fully adapts domain-specific AOM applications, there is still need for research in matters such as inheritance and behavior representation. The analysis of factors such as reuse cost, regarding quality and productivity, were also left for future works. The authors are currently looking for large or medium scale AOM applications where this model can be applied.

We thank for the essential support of FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) to this research.

References

[Arsanjani 2001] Arsanjani, A.: “Rule Object: a pattern language for adaptive and scalable business rule construction”; Proc. of 8th Conference on Pattern Languages of Programs (PLoP) (2001).

[Acherkan et al. 2011] Acherkan, E., Hen-Tov, A., Lorenz, D. H., Schachter, L.: "The ink language meta-metamodel for adaptive object-model frameworks"; in Proc. of 26th ACM International Conference Companion on OOPSLA Companion (2011).

[Bhati 2009] Bhati, S.: "Applying Adaptive Object Model using Dynamic languages and Schema-less Databases", Posted at 16 Nov 2009, Accessed at 12/11/2013 on http://weblog.plexobject.com/?p=1667.

[Ferreira et al. 2010a] Ferreira, H. S., Correia, F. F., Aguiar, A., Faria, J. P.: "Adaptive Object-Models: a research roadmap", in International Journal on Advance in Software, 3, 1 (2010) 70-89.

[Ferreira et al. 2010b] Ferreira, H. S., Correia, F. F., Yoder, J., Aguiar, A: "Core patterns of object-oriented meta-architectures"; in Proc. of 17th Conference on Pattern Languages of Programs (PLoP) (2010).

[Fowler 1996] Fowler, M.: "Analysis patterns: reusable object models"; Addison-Wesley Professional (1996).

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns: elements of reusable object oriented software"; Addison-Wesley (1994).

[Garud et al. 2008] Garud, R., Jain, S., Tuertscher, P.: "Incomplete by design and designing for incompleteness," in Organization studies as a science of design, 29, 3 (2008) 351-371.

[Guerra 2012] Guerra E. M. et al., "Projeto Esfinge" available on: http://esfinge.sf.net/, accessed in: 12 apr. 2012.

[Guerra et al. 2013a] Guerra, E., Souza, J. T., Fernandes, C.: " Pattern Language for the Internal Structure of Metadata-based Frameworks" in Transactions on Pattern Languages of Programming, 3 (2013) 55-110.

[Guerra et al. 2013b] Guerra, E., Buarque, E., Fernandes, C., Silveira, F.: "A Flexible Model for Crosscutting Metadata-Based Frameworks"; in Lecture Notes in Computer Science, Computational Science and Its Applications – ICCSA 2013, 7972 (2013) 391-407.

[Guerra 2014] Guerra E. M., "Designing a Framework with TDD: A Journey". IEEE Software, v. Jan/Fe, p. 9-14, 2014.

[Ferreira 2010] Ferreira, H. S.: "Adaptive-Object Modeling: Patterns, Tools and Applications"; PhD Thesis, Faculdade de Engenharia da Universidade do Porto (2010).

[Ferreira et al. 2009] Ferreira, H. S., Correia, F. F., Aguiar, A.: "Design for an Adaptive Object-Model framework: an overview"; Proc. of 4th Workshop on Models@Run.Time (2009).

[Hen-Tov et al. 2009] Hen-Tov, A., Lorenz, D. H., Pinhasi, A., Schachter, L. : "ModelTalk: when everything is a domain-specific language", in IEEE Software, 26, 4 (2009) 39-46.

[Johnson and Wolf 1997] Johnson, R., Wolf, B.: "Type Object," in Pattern Languages of Program Design 3, Addison-Wesley (1997) 47-65.

[Matsumoto and Guerra 2012] Matsumoto, P. ; Guerra, E. M.: "An Architectural Model for Adapting Domain-Specific AOM Applications". In: SBCARS- Simpósio Brasileiro de Componentes, Arquitetura e Reutilizacação de Software, 2012, Natal.

[Membrey et al. 2010] Membrey, P., Plugge, E., Hawkins, T.: "The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing"; Apress (2010).

[Riehle et al. 2000] Riehle, D., Tilman, M., Johnson, R.: "Dynamic Object Model"; Proc. of 7th Conference on Pattern Languages of Programs (PLoP) (2000).

[Silva et al. 2013] Silva, J., Guerra, E., Fernandes, C.: "An Extensible and Decoupled Architectural Model for Authorization Frameworks"; in Lecture Notes in Computer Science, Computational Science and Its Applications – ICCSA 2013, 7974 (2013) 614-628.

[Steward 1981] Steward, D. V.: "The Design structure system: A method for managing the design of complex systems", IEEE Transactions on Engineering Management, vol. 28 (1981) pp. 71-74.

[Welicki et al. 2007a] Welicki, L., Yoder, J. W., Wirfs-Brock, R., Johnson,R. E.: "Towards a pattern language for Adaptive Object-Models" Proc. of 22th Object-Oriented Programming, Systems, Languages & Applications (2007).

[Welicki et al. 2007b] Welicki, L., Yoder, J. W., Wirfs-Brock, R.: "A pattern language for Adaptive Object Models - rendering patterns"; Proc. of 14th Conference on Pattern Languages of Programs (PLoP) (2007).

[Yassine 2004] Yassine, A. A.: "An introduction to modeling and analyzing complex product development processes using the Design Structure Matrix (DSM) method"; in Quaderni di Management (Italian Management Review), 9 (2004).

[Yoder et al. 2001] Yoder, J. W., Balaguer, F., Johnson, R.: "Architecture and design of Adaptive Object-Models"; In Proceedings of the 16th Object-Oriented Programming, Systems, Languages & Applications (2001).

[Yoder, Johnson 2002] Yoder, J. W., Johnson, R.: "The Adaptive Object-Model architectural style"; Proc. of 3rd IEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (2002).